F. Dignum, E. Verharen and S. Bos. Implementation of a cooperative agent architecture based on the language-action perspective. In M. Singh et.al., editor, *Intelligent Agents IV (LNAI 1365)*, pages 31-44, Springer Verlag, 1998.

Implementation of a Cooperative Agent Architecture based on the Language-Action Perspective

Egon Verharen[*] Frank Dignum[†] Sander Bos[‡]

[*] Infolab, Tilburg University
Tilburg, POBox 90153, 5000 LE, the Netherlands
E.M.Verharen@kub.nl

[† ‡] Fac. of Maths & Comp. Sc., Eindhoven University of Technology
Eindhoven, POBox 513, 5600 MB, the Netherlands
dignum@win.tue.nl,A.J.Bos@stud.tue.nl

Abstract. In this paper the architecture and implementation of Cooperative Information Agents (CIA) is described. Taking a language-action perspective to the design of CIAs allows for the specification of obligations and authorizations, and results in the separation of tasks (things the agent must do) and contracts (mutually agreed commitments to the course of communication). The architecture describes the functional components of a CIA: task manager (responsible for managing the agenda), contract manager (managing and negotiating contracts), communication manager (responsible for all external communication), and service execution manager (managing the execution of actions). The prototype agents show how a formal logical theory for communicating agents can be used as a sound basis for an actual implementation.

1 Introduction

Traditionally an information system (IS) was considered as one central database and a set of users accessing the database through application programs or directly via an SQL interface. Today, ISs are connected to each other and have to be accessible using electronic networks and EDI, while still maintaining their autonomy. Complete integration of the various resources might not be possible for technical or organizational reasons, hence the growing reliance on interaction between systems. This led to the paradigm of cooperative information systems (CIS) introduced in [14]. For systems to be able to cooperate they must have an intelligent interface that can cope with all types of requests and eventualities. A CIS actively maintains its information; it can communicate with other systems and reason about the information that it contains. It might decide to search for information that it needs by inquiring for it from other CISs if it knows that it does not contain the information itself, preferably in ways it negotiates with (and lays down in contracts with) those other systems. It can respond more intelligently to messages explaining why a request does not have an answer, or propose alternatives. And it can ne-

gotiate about which requests it is willing to respond to and which requests will have no effect. For this purpose the CIS should contain a task module that plans the tasks the CIS has to fulfill. A *task* is a meaningful unit of work assigned to an agent. Performing the task often involves initiating communication transactions. However, the task's specification and updates thereof do concern the agent in question only, whereas changes in the possible transactions (involving other agents) can only be made by consent of those agents. For that reason, we make a distinction between task and contract, where the contract corresponds to the agreements between agents and the task draws on this potential for fulfilling an agent's goal. We refer to an autonomous CIS with tasks and contracts as a *Cooperative Information Agent* (CIA).

In our approach we place much more emphasis on the communication and negotiation between CIAs than takes place in occasional contacts. Because the abilities of the CIA to communicate and negotiate take an important place we claim that the influence of *linguistics* for these systems should go beyond that of a natural language interface. We use a language-action perspective [10] (based on the speech act theory as developed by Searle [15, 16] and Habermas [12]) to describe the communication itself and guide the architecture. In contrast to traditional data-flows the language-action perspective emphasizes what agents (human or automated) *do* while communicating and how communication brings about a coordination of their activities. The focus is on language as action, and the speech act is the basic unit of communication. We view a CIA as a kind of normative system, in which the coordination of activities is governed by making commitments, described by obligations and authorizations of the communicating agents. An obligation is the result of a commitment to perform a certain act and authorizations restrain or allow the commitment to and operation of an act (including doing other communicative acts). Because a CIA must be able to reason about its tasks, contractual obligations and the information that it possesses we think it is crucial that there is an underlying formal theory in which the agents can be described (including the communication). We have described this theory in a multi- modal logic (see e.g. [5, 19, 20]). In this paper we focus on the way this theory is used, in concepts such as authorization relationships between agents and how obligations (resulting from tasks and contracts) are dealt with. The agents do not contain theorem provers that derive new knowledge from the knowledge they possess and the communication they have. However, the way that the agents act is conform to the axioms that hold for the underlying logic.

This paper describes a (conceptual) architecture for CIAs, that integrates much of our previous work ([20, 6, 7]), and its implementation. The structure of this paper is as follows. In section 2 the CIA architecture is presented. In section 3 its implementation is described. Section 4 presents an example of a CIA specification and the resulting agent, and section 5 gives some conclusions and areas for further research.

2 Architecture

In figure 1 we show the (global) architecture of a CIA (its main functional modules and their interaction) as we use it. Although the components are similar to the ones in other agent architectures described in the literature (e.g., the general architecture of a social agent in [13] and the TAEMS architecture [3]) both the structure and working of the

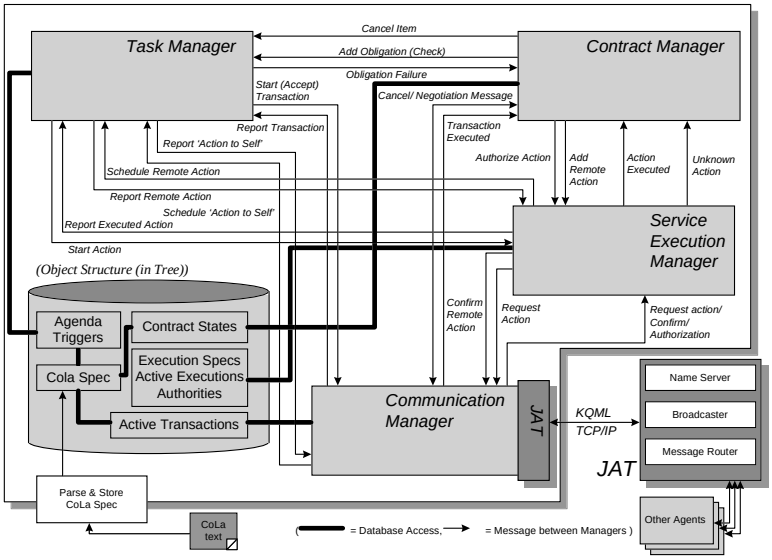

Fig. 1. General CIA architecture

components, based on our communication-oriented approach, is distinct. In the rest of this section we will give a short overview of the most important features of the components in this architecture.

The CIA consists of a number of functional modules, each responsible for one of the agents' main activities: task management, contract management, communication and interaction, and execution of services.

Each agent has an *agenda* containing the actions to be performed by the agent, instantly or at some designated time. The agenda is derived from the *obligations* of the agent. The agent can add new items to the agenda (typically done on the request of another agent) and can reason about them. Obligations can be the result of the (sub)task of the agent, but can also follow from the contract (see below). Items can be removed from the agenda by performing actions or by violating an obligation. In the latter case usually some compensatory action has to be performed. Maintaining the agenda is the primary task of the task manager, which therefore is the central module of the agent.

The pro-active behavior of a CIA is determined by the tasks the CIA defined in *CoLa* (Communication and Coordination Language) during the specification of the agent.

The agent performs *actions* and *transactions*. Actions can be performed by the agent itself (executed and monitored by the *service execution manager*) without interaction with other agents. Transactions are actions that involve communication with other agents. The basic building blocks of transactions are the speech acts. Agents communicate with each other by sending KQML messages. The communication is initiated and monitored by the *communication manager*. Transactions are used in a *contract* describing the commitments between two agents. The contract also specifies what should happen in case of a violation of one of the obligations, or cancellation by one of the agents, possibly leading to other obligations described by another transaction in the contract, and triggering a

”contingency” plan, describing what should happen in order to get the subtask fulfilled. Both agents have direct access to the contracts between them. The *contract manager* monitors the contracts that a CIA is involved in and decides what steps to take when a contract is breached.

In the next subsections the working of the different managers is described in more detail.

2.1 Task Manager

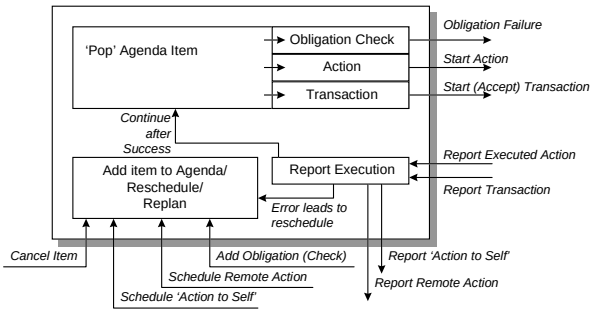

Fig. 2. Task Manager

The *task manager*'s main function is managing the agenda. The execution model of the task manager is as follows: when a task is called or a goal is established, the Task Manager devises a plan to fulfill the goal or perform the task. The plan consists of a number of subtasks with precedence relations and alternatives (see section 3.3). The subtasks are put on the agenda in the right order (the task manager schedules the items on the agenda in an order based on their constraints and deadlines). It then tries to perform all subtasks, backtracking when a task fails or an exception occurs. In case of an action the service execution manager is notified, in case of a transaction control is transferred to the communication manager. A special kind of action is the obligation check in which the task manager evaluates the knowledge bases (history) to see if another agent has fulfilled its obligation. If the check fails the contract manager is notified.

Special attention should be paid to cancellations of both actions and transactions. In case of a cancellation a contingency plan, that can be specified separately, can be triggered. A notorious problem with contingencies is that later (dependent) subtasks may already have completed, but their result have become obsolete. Whether they have to be retried or not depends on what kinds of results they have produced. A contingency plan consists of a set of *results* that come about by certain subtasks and can become invalidated by other subtasks. When this occurs, a task can be triggered to repair the damage. This task can make use of the fact that all the essential results obtained so far (and not invalidated) are explicit. E.g., if a hotel-reservation is dependent on an airline-reservation, and the flight is canceled, the contingency plan can try to repair the damage by trying another airline. Only if that fails the hotel- reservation has to be canceled (independently, there can be a sanction on the party canceling, as specified in the contract).

2.2 Communication Manager

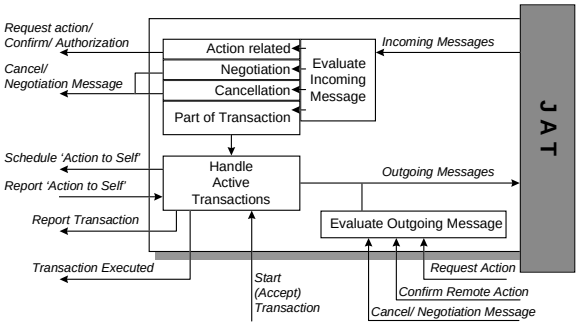

Fig. 3. Communication Manager

The *communication manager* handles all external communication of a CIA. When it receives a request for executing a transaction it constructs a communication plan (based on possible deadlines and constraints on the order of messages specified in the transaction). It handles all incoming messages, routing them to the appropriate managers.

We do not assume that the CIAs follow a fixed communication protocol. Therefore we need a rich communication language in which also the intent of each message is clear. This is achieved by basing the messages on the theory of speech acts (Searle [15, 16]). Using speech acts we can model existing protocols that are often used, e.g., the Contract Net Protocol, or the protocol from the ADEPT framework [18] as is done in [4]. However, the CIA can also react when other agents do not follow the same protocol. Communication through the Java Agent Template (JAT) takes place by sending KQML packages. However, we only use our own defined KQML performatives for our communication language which have a clearly defined semantics in the underlying logic (unlike some standard KQML performatives (see also [2] and the FIPA proposal [9])).

2.3 Service Execution Manager

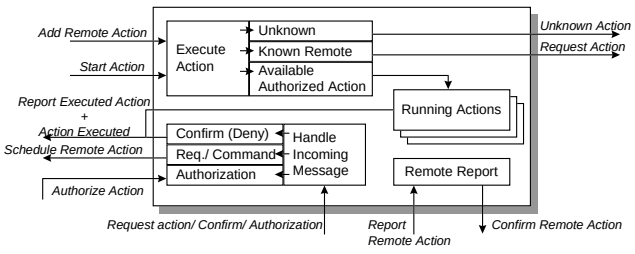

Fig. 4. Service Execution Manager

The *service execution manager (SEM)* manages the local actions. It may also handle simple exceptions within the services (usually when the service can be restarted). The SEM also manages the services the CIA can give to other CIAs. I.e., it checks whether the other agents are authorized to request the service. It maintains a database with information about services the CIA can provide itself and also services provided by other CIAs. In this way it assists the task manager in the execution of tasks that the CIA cannot perform itself but has to request from others.

The SEM distinguishes three authorization relationships: *power* (which is a fixed institutional relationship, allowing an agent to command the execution of a service); *authority* (which must be granted explicitly and disappears after use or some time. E.g. authority to demand payment for delivered products); and *charity* (or *peer* relationship, where an agent can only request the execution of a service and no guarantee is given on the execution). If no relationship with the other agent is recorded, the contract manager is notified to negotiate the execution of the service.

2.4 Contract Manager

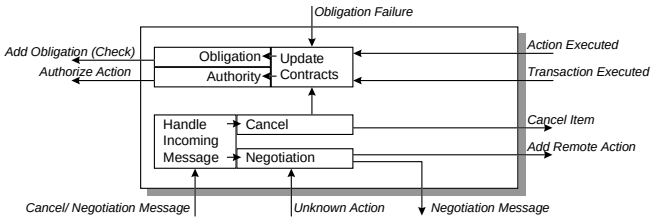

Fig. 5. Contract Manager

The agreements about the interactions between the CIAs are described in the contracts. In our framework, contracts conceptually specify obligations between different parties about services provided to each other. If a particular service is not being fulfilled, it is possible to reason about this violation and take a remedial action without forcing the whole task to abort. A contract describes the authorized communication behavior among providers and receivers of services. If the provider does not adhere to the obligation it is the job of the contract manager to impose the violation policies.

The contract manager also negotiates contracts with other agents. At the moment only a simple form of Contract Net negotiation is implemented, but other negotiation strategies can be incorporated.

Contracts are specified by Petri-nets. If interaction proceeds as planned the contract manager only notifies the task manager to add obligation checks to the agenda. In case of a violation (or any other event that causes the entrance of a failure place in the contract, e.g. in case of a cancellation) the contract is examined and the task manager is notified to add the appropriate obligations to the agenda.

3 Implementation

In this section some of the aspects of implementation are described. Several sources of information are used by all parts of the CIA. The loaded CoLa specification is stored as a whole in an object-structure with a schema similar to the CoLa specification. This object-structure acts as input to and reference for the rest of the agent. The 'real-world knowledge' of the agent and the history of events, which is important for determining the truth-value of conditions and deadlines, is also used throughout the CIA. This data is stored as a number of linked objects in a list.

3.1 Java Agent Template

The basis of the CIA implementation is the Java Agent Template (JAT) [11]. The JAT offers basic agent functionality, its agents are non-mobile and can exchange any type of information. The CIA is running as an interpreter (message-handler) in the JAT, and is initialized when the JAT starts.

The CIA Implementation uses only the low-level (KQML) communication of the JAT, not the more advanced features. Sending and receiving of messages is done through just two primitives in the CIA, making it easy to replace the JAT with another communication system.

The central router mostly serves as an 'agent-name to address'-list maintainer, and accepts only sign-on/ sign-off messages, Internet address-queries based on agent names and requests for broadcasting a message. There is no such thing as a 'Facilitator' (see e.g. [17]) in our implementation to support the agents functionality, the agents are mostly truly autonomous.

3.2 Tableau Decomposition

As described in e.g. [19, 1], the order in which 'action-items' (transactions, messages, local actions) in the CIA may take place are specified using constraints in Propositional Temporal Logic. This logic extends the simple Propositional Logic by adding the operators SOMETIMES, NEXT, ALWAYS and BEFORE.

A graph is created of all possible paths of item orderings allowed by the constraints. To determine this dependency graph the Tableau Decomposition method is used, which is explained in detail in [1]. A small and contrived example of a dependency graph for three items ('a', 'b' and 'c'), and two constraints on those items,

```
a BEFORE (b OR c), b BEFORE c
```

is shown in figure 6. The constraints that identify the nodes are placed below them. The created graphs are used in two different ways in the CIA. The task manager uses them to create the agenda and the communication manager uses them to determine which actions can be performed immediately and which actions must wait. This is explained in more detail in the next sections where some implementation issues of these modules are described.

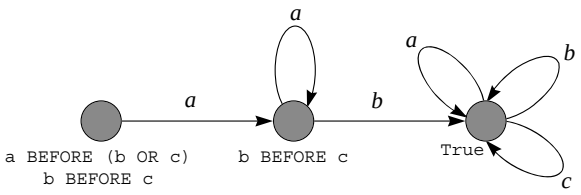

Fig. 6. A small example of a dependency graph

3.3 Task Manager

The information stored by the task manager all relates to the Agenda, which formally stores 'the obligations of the agent, a set of deontic temporal constraints'. Informally, the agenda specifies everything the agent has to do (and when). In practice, there are three types of items on the agenda: Actions, transactions and obligation checks. The obligation checks are performed for the contract manager. Actions and transactions may originate from tasks, local actions in transactions, violations of contracts and authorized actions executed for other agents. So, even though local actions are executed by the execution manager they are also reflected on the agenda!

All these items may have a deadline specified, which is used to order the agenda-items. All items are scheduled in the order of their deadlines, with the additional constraint that all items related to one task must be placed in their original order (the order of the current plan for that task). When an item of a task is being executed the rest of the items of the same task are 'frozen'. All unfrozen items on the agenda may be executed, which means that items placed below frozen items may be executed immediately.

When a task is called a complete path leading to the (specified) goal is selected in the plan graph created for the task. If a path in the graph ends without reaching a goal the path is backtracked looking for an alternative path which ends in a goal-state. Once a path leading to a goal has been found the items on this path are placed on the agenda in sequence. If execution of one of the items fails it might be possible to find another path using an alternative in the graph for the failed item, or to backtrack from that point looking for a new path to a goal. Contingency actions may be specified for executed items that have become obsolete because of backtracking.

There is also a lists of triggers. Triggers are items which are waiting for events, after which they perform some kind of action (on the agenda). Deadlines may be specified for the triggers. There are different kinds of triggers:

- A trigger is created to handle the report send by managers when they have finished processing an action/ transaction. Activation of the trigger may lead to either routing the report to another manager that was responsible for placing the item on the agenda, unfreezing the remainder of a task or causing a replan of the task (if the item failed).
- Obligation checks must not be placed on the agenda immediately, but only when the obligation's deadline has passed.
- Tasks may have a precondition, in which case they will only be scheduled when the precondition is activated.

3.4 Communication Manager

The communication manager uses the graphs created from the transaction specifications to determine the possible entries of communication. When a transaction starts the communication manager examines which messages (and local actions) may be executed initially, by examining a path leading to a desired state. All the items that can be reached on a path to a goal without entering messages which are send by the partner in the transaction (the plan-graph also stores which side initiates each item) or a local action (which has to be completed before the transaction may proceed) may be selected and executed. When an agent receives new messages related to an active transaction (or a local action related to an active transaction is completed) it recreates the path in the plan-graph from the history of the transaction. If the path cannot be recreated, the other agent did not keep to the constraints and the transaction fails. Otherwise it may be that the agent can continue the transaction based on the newly added events (if it is allowed to select new items on a continuing path to the goal) or that it must wait. Because the execution of local actions and the transmitting of messages is final, backtracking is not used for the transaction graphs.

For each running transaction information such as a reference to the specification and plan graph, instantiated deadlines and a message history is stored.

When messages are part of a transaction the transaction-name and a unique identifier are put in each message in order to be able to determine the 'conversation' to which they belong. (Not all messages are part of transactions! The most notable are cancellations.) When a transaction starts, one agent must initiate the transaction and one agent must accept it. To alleviate the synchronization of this process, messages belonging to unresolved transactions are buffered for a limited amount of time at the receiving end, waiting for the task manager to deliver the corresponding acceptance.

4 Example

To give an idea of the actual CIA-implementation an example specification for an agent is given concerning the well-known case of booking a business trip [8]. We also included some screenshot of the running agent that results from this specification.

4.1 Example Specification

This section describes some of the more important fragments of the CoLa specification used for the travel agent. The complete specification of all four agents of the business-trip example can be found on the CIA WWW-page:

`http://machtig.kub.nl/egon/CIA/`

In the business-trip example there are four (types of) agents, the user, the travel- agent, the airline and the hotel. We will focus on the travel agent here, which communicates with all the other agents. The travel agent is called upon by the user agent with the request for a trip. The travel agent then books a flight and reserves a hotel based on the user's wishes. The accompanying CoLa specification consists of three parts, the tasks of the travel agent, specifications of the transactions and specification of the contracts.

Travel Agent Tasks The main task of the travel agent, 'plan trip', consists of accepting a request from the user, booking the flight and the hotel, notifying the user of the arrangements and making sure the user pays.

```
task plantrip
subtasks:  accept(booktrip);
           T.flightreservation(klm);
           hotelreservation;
           T.bookedtrip(trip);
           T.payment;
```

There are several constraints on the order in which these actions may be performed; initially the user will ask the travel agent to generate a trip through a 'booktrip'-transaction, while finally the user must pay for his trip.

```
constraints:
      accept(booktrip) BEFORE
          (T.flightreservation(klm) OR hotelreservation);
      T.flightreservation(klm) BEFORE T.bookedtrip(trip);
      hotelreservation(klm) BEFORE T.bookedtrip(trip);
      T.bookedtrip(trip) BEFORE T.payment;
goal =   T.payment
```

During execution of the task several results are created. The task-specification can contain information on how results are created, and what events have what effect on them. An example:

```
dependencies: "trip" depends-on "ticket"
                                  (create fail update);
contingency:  result "ticket" created-by plantrip
                 closed-by  T.betaling
                 updated-by T.changeflight
                 invalidated-by A.skip_it
                 compensated-by A.removedb(ticket)
              end-result;
```

In the 'plantrip' task 'hotelreservation' is defined as a sub-task, which must reserve a room at the Sheraton or the Hilton (with a preference for the Sheraton).

```
task hotelreservation
subtasks: T.hotelreserve(sheraton);
T.hotelreserve(hilton);
goal = T.hotelreserve(sheraton) [2] XOR
       T.hotelreserve(hilton)   [1]
end-task;
```

Travel Agent Transactions One of the simpler transactions is the one which initiates the communication between the agents. The user requests the travel agent to reserve a trip, the travel agent then either starts a new reservation and sends a confirmation or refuses.

```
transaction booktrip
agents:  t: travelagent;  u: user;
         t can send messages:
                A.reserve(trip) to self;
                confirm(A.reserve(trip)) to u;
                assert(refuse-to(A.reserve(trip))) to u;
         u can send messages:
                request(A.reserve(trip)) to t;
         constraints:
                request(A.reserve(trip)) BEFORE
                  (A.reserve(trip) XOR
                    assert(refuse-to(A.reserve(trip))));
                A.reserve(trip) BEFORE
                    confirm(A.reserve(trip));
goal = confirm(A.reserve(trip))
exit = assert(refuse-to(A.reserve(trip)))
end-transaction;
```

Travel Agent Contracts One of the contracts specified is used between the travel agent and the airline. It consists of three states. The contract is activated once the 'flightreservation' transaction is completed. Then, if the flight is performed the contract reaches the acceptance state, but if the flight is canceled a fine must be paid first.

```
  contract flight
  agents:   a: airline;  t: travelagent;
  clauses:
        S1: obl(a, A.fly)
        in: T.flightreservation(klm)
        goal: A.fly => S2
        exit: cancel("flight") => S3
        end S1;

        S2: acc(t) in: A.fly, T.payfine end S2;

        S3: obl(a, T.payfine)
        in: cancel("flight")
        deadline: T.payfine << NOW + 14 days
        goal: T.payfine => S2
        end S3;
  end-contract;
```

The above specification is a textual representation of the Petri-net definition of the contract, which is omitted due to space limitations.

4.2 Screenshot running agent

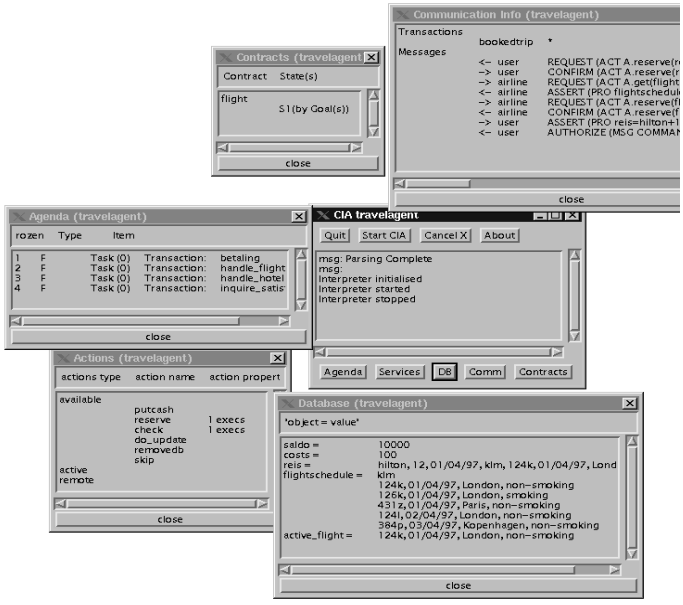

Fig. 7. Screenshot of example CIA

In figure 7 a screenshot can be seen of the actual CIA- implementation. At the center right the main control window of the travel agent is shown, with display windows for the agent's state around it.

The messages in the main window show that the travel agent has parsed its specification and initialized correctly, and has been started and stopped. The buttons at the top of the window allow for several general actions (one of which is the unprepared cancellation of realized actions and transactions), while the buttons at the bottom allow the user to view several aspects of the state of the agent. All of these windows have been opened in the screenshot.

The state-windows show that the agent has been running for a while. All items placed on the agenda (shown left) belong to the same task and are all frozen. This is because the travel agent is currently working on the transaction 'booked trip' of the same task, as can be seen in the communication window (top right). The communication window also shows that the travel agent has send and received messages to and from the user agent and the airline, according to several executed transactions. For instance, in the fourth message the airline has asserted a flight- schedule to the travel agent. This flight-schedule

has since been stored in the travel agent's database (bottom right), in which you can also see the trip[1] the travel agent has created for the user (in the example, the hotel-reservation has been created by the travel agent itself). The flight has been selected by a local 'check' action, which is one of the actions executed by the agent (the 'Actions'-window at the bottom left). In the 'Contracts'-window (top middle) one can see the flight-contract is still in its initial state.

5 Conclusion

In this paper we have given an impression of the architecture and implementation of Co-operative Information Agents based on the language-action perspective. Important elements that come back in the implementations are the use of speech act theory for the communication between agents and especially the commitments and obligations following from the communication. Another important aspect from the agent architecture is the distinction between tasks and contracts. This distinction gives advantages in the case of failure of tasks or cancellations of results.

Future research should include the scalability of this prototype. Also some aspects such as the negotiation protocol and conversation rules are now only present in their most simple form and should be upgraded to more realistic formats.

References

1. P. Cohen and H. Levesque. Communicative actions for artificial agents. In *Int. Conf. on Multi-Agent Systems*, pages 65–72, 1995.
2. K. Decker and V. Lesser. Task environment centered design of organizations. In *AAAI Spring Symp. on Computational Organization Design*. Stanford, 1994.
3. F. Dignum. Social interactions of autonomous agents: Private and global views on communication. In P.-Y. Schobbens, editor, *Proc. of 3rd workshop of the ModelAge Project*, Siena, Italy, 1997.
4. F. Dignum and B. van Linder. Modeling rational agents in a dynamic environment: Putting humpty dumpty together again. In J.L. Fiadeiro and P.-Y. Schobbens, editors, *Proc. of 2nd workshop of the ModelAge Project*, pages 81–92, Sesimbra, Portugal, 1996.
5. F. Dignum and H. Weigand. Communication and deontic logic. In R. Wieringa and R. Feenstra, editors, *Information Systems, Correctness and Reusability*, pages 242–260, Singapore, 1995. World Scientific.
6. F. Dignum and H. Weigand. Modeling communication between cooperative systems. In J. Iivari et. al., editor, *Proc. of CAISE'95*, pages 140–153, Berlin, 1995. Springer-Verlag.
7. A. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufman, 1992.
8. P. Faratin T. Norman, N. Jennings and E. Mamdani. Designing and implementing a multi-agent architecture for business process management. In M. Wooldridge J. Mueller and N. Jennings, editors, *Intelligent Agents III - Proceedings ATAL-96*, pages 149–162, 1996.
9. Fipa. http://drogo.cselt.stet.it/fipa
10. F. Flores and J.J. Ludlow. Doing and speaking in the office. In G. Fick et. al., editor, *DSS: Issues and Challenges*, pages 95–118, New York, 1980. Pergamon Press.

[1] 'reis' means trip in the Dutch language

11. R. Frost. The jat. http://cdr.stanford.edu/ABE/JavaAgent.html
12. J. Habermas. *The Theory of Communicative Action: Reason and the Rationalization of Society, Volume One*. Beacon Press, Boston, 1984.
13. D. McKay T. Finin, R. Fritzson and R. McEntire. Kqml: a language and protocol for knowledge and information exchange. In M. Klein et. al., editor, *Distributed AI - 13th Int.l. WS*, pages 99–103, Menlo Park, CA., 1994. AAAI Press.
14. R.A. Meersman A.H.H. Ngu and H. Weigand. Specification and verification of communication for interoperable transactions. In *Int.l. Journal of Intelligent and Cooperative Information Systems, vol. 3, no. 1*, pages 47–65, 1994.
15. B. Moulin and B. Chaib-draa. An overview of distributed artificial intelligence. In H. O'Hare and N. Jennings, editors, *Foundations of DAI*, pages 3–55, New York, 1996. John Wiley and Sons Inc.
16. M. Papazoglou. *An organizational framework for intelligent cooperative IS*. IJICIS-1(1), 1992.
17. J.R. Searle. *Speech Acts*. Cambridge University Press, 1969.
18. J.R. Searle and D. Vanderveken. *Foundations of illocutionary logic*. Cambridge University Press, 1985.
19. E.M. Verharen. *A Language-Action Perspective on the Design of Cooperative Information Agents*. PhD thesis, Katholieke Universiteit Brabant, 1997.
20. E.M. Verharen and F. Dignum. Cooperative information agents and communication. In M. Klusch, editor, *Proc. of the 1st Int.l. WS on CIAs*, Berlin, 1997. Springer-Verlag.

This article was processed using the LaTeX macro package with LLNCS style